MapReduce & Hadoop

Why is MapReduce?

- By definition, big data is too large to handle by conventional means. Sooner or later, you just can't scale **up** anymore
  - machines cannot grow too large…
- But we can add more computers!
- MapReduce paradigm allows us to scale **out**
  - even with commodity hardware
  - first proposed in the big data world by Google

Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* 2004.

What is MapReduce?

- A programming paradigm (for big data processing)
  - Data is split into distributable chunks
  - Perform a series of transformations on those chunks
  - Transformations are run in parallel on the chunks (data parallelism)
- MapReduce is scalable by adding more machines to process chunks
  - scaling out on commodity hardware
- The foundation for Hadoop, which is an implementation of MapReduce

What is MapReduce?

- A programming paradigm that process data in 2 phases/operations: **map()**, and then **reduce()**
  - instead of map(), filter(), reduce() in Python, we are limited to only map() and reduce()
- In a nutshell — *that's it*:
  - Provide a data collection (separable records)
  - Apply a user-defined map function on each data record
  - Then reduce the mapped output with another user-defined function

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

INPUT: list of key-value pairs of (k1,v1)

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

  INPUT: list of key-value pairs of (k1,v1)

  MAP: (k1,v1) ➔ [list of (k2,v2)]

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

  INPUT: list of key-value pairs of (k1,v1)

  MAP: (k1,v1) ➔ [list of (k2,v2)]

  SHUFFLE: combine (k2,v2) ➔ (k2, [list of v2])

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

  INPUT: list of key-value pairs of (k1,v1)

  MAP: (k1,v1) ➔ [list of (k2,v2)]

  SHUFFLE: combine (k2,v2) ➔ (k2, [list of v2])

  REDUCE: (k2, [list of v2]) ➔ (k3,v3)

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

  INPUT: list of key-value pairs of (k1,v1)

  MAP: (k1,v1) ➔ [list of (k2,v2)]

  SHUFFLE: combine (k2,v2) ➔ (k2, [list of v2])

  REDUCE: (k2, [list of v2]) ➔ (k3,v3)

  OUTPUT: list of (k3,v3)

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

Fixed pipeline

INPUT: list of key-value pairs of (k1,v1)

MAP: (k1,v1) ➔ [list of (k2,v2)]

SHUFFLE: combine (k2,v2) ➔ (k2, [list of v2])

REDUCE: (k2, [list of v2]) ➔ (k3,v3)

OUTPUT: list of (k3,v3)

MapReduce Pipeline

- MapReduce works on ***(key,value) pairs***

INPUT: list of key-value pairs of (k1,v1)

MAP: (k1,v1) ➔ [list of (k2,v2)]

SHUFFLE: combine (k2,v2) ➔ (k2, [list of v2])

REDUCE: (k2, [list of v2]) ➔ (k3,v3)

OUTPUT: list of (k3,v3)

User-defined functions
executed in parallel

Input

- Data must be separable into records
  - Lines of text
  - Rows of tables
  - CSV: ***yes*** — JSON, XML: ***no*** (except independent JSONs/XMLs)
- Key/value pairs
  - Key = line number, record index
  - Value = text string, row data
- Keys are mostly ignored in many cases (e.g. just passing text lines)

Map phase

- Transform each input record using a user-defined function
  - **(k1,v1)** ➔ [**(k2,v2)**,…] — *could be an empty list*
- one-to-many relationship
  - vs. one-to-one of Python's map() higher order function
  - Python's filter() is part of the Map phase
- Mainly process by streaming data chunks
  - [**(k1,v1)**] ➔ [**(k2,v2)**,…]
  - *generators* ➔ *generators*

Shuffle & Sort Phase

- For each key **k2**, collect all **v2** from outputs of all map processes — *shuffling* into **(k2, [v2])**
  - Done by distributed partitioning and local sort
- Outputs are **(k2, [v2])** being distributed to “reducers”
- [**(k2,v2)**] ➔ **(k2, [v2])**
- **(k2,v2)** are *intermediate* key/value pairs

Reduce phase

- Transform the output of the Shuffle phase using a user-defined function
  - **(k2,[v2])** ➔ **(k3, v3)**
- Equivalent to reduceByKey() ~ groupByKey() x reduce()
  - vs. Python's reduce()
- Also process in streaming data chunks
  - [**(k2, [v2])**] ➔ [**(k3, v3)**]
- This phase is *optional*, can be omitted.

Output

- Redirect output from each reducer as an unsorted list of **(k3, v3)**
  - Store as separate files
  - Print to the console
  - Load data into databases

The classic example: WordCount

The classic example: WordCount

- The “Hello World” of MapReduce

The classic example: WordCount

- The “Hello World” of MapReduce
- Given a set of documents, compute the frequency of each word

The classic example: WordCount

- The “Hello World” of MapReduce
- Given a set of documents, compute the frequency of each word
- INPUT **(k1, v1)**: (document:line number, text line)

The classic example: WordCount

- The “Hello World” of MapReduce
- Given a set of documents, compute the frequency of each word
- INPUT **(k1, v1)**: (document:line number, text line)
- OUTPUT **(k3, v3)**: (word, frequency)

The classic example: WordCount

- The “Hello World” of MapReduce
- Given a set of documents, compute the frequency of each word
- INPUT **(k1, v1)**: (document:line number, text line)
- OUTPUT **(k3, v3)**: (word, frequency)
- INTERMEDIATE **(k2, v2)**: (word, 1)

The classic example: WordCount

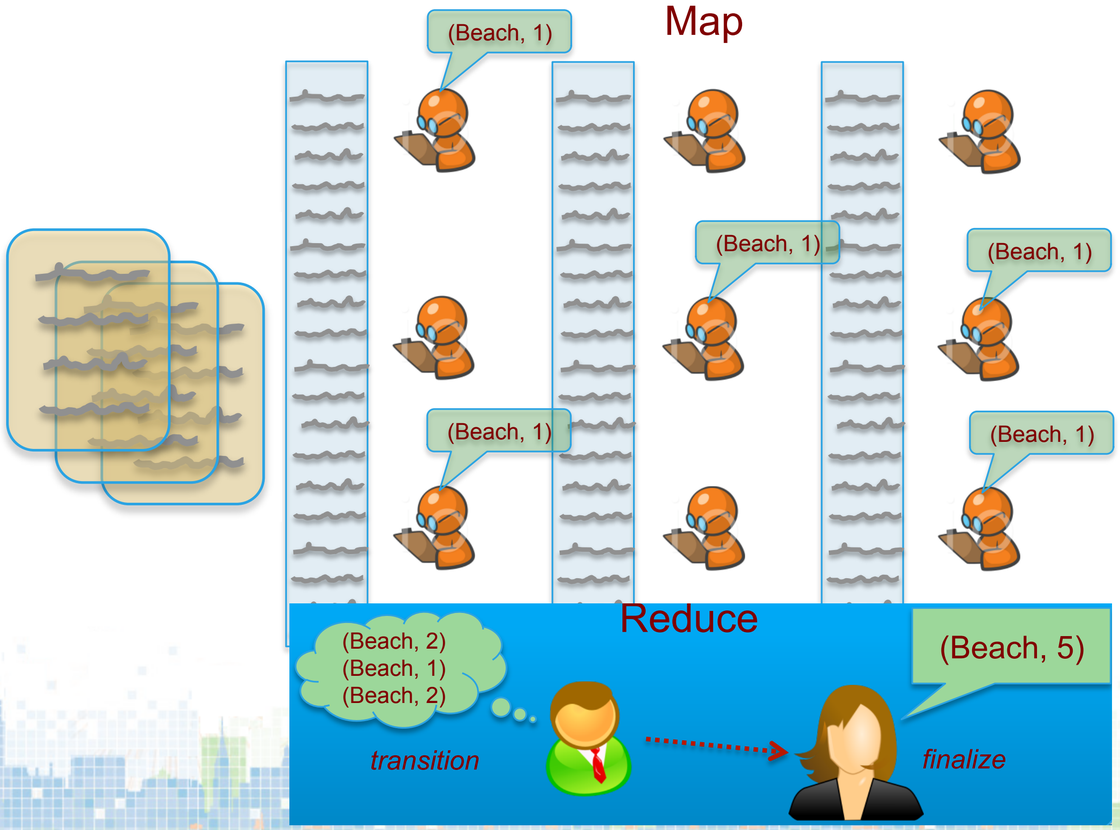
- The “Hello World” of MapReduce
- Given a set of documents, compute the frequency of each word
- INPUT **(k1, v1)**: (document:line number, text line)
- OUTPUT **(k3, v3)**: (word, frequency)
- INTERMEDIATE **(k2, v2)**: (word, 1)
  - Transform each line to a list of words and their frequencies (=1)

The classic example: WordCount

- The “Hello World” of MapReduce
- Given a set of documents, compute the frequency of each word
- INPUT **(k1, v1)**: (document:line number, text line)
- OUTPUT **(k3, v3)**: (word, frequency)
- INTERMEDIATE **(k2, v2)**: (word, 1)
  - Transform each line to a list of words and their frequencies (=1)
  - Combine all tuples by words, and add all the frequency together

The classic example: WordCount

- The “Hello World” of MapReduce
- Given a set of documents, compute the frequency of each word



<1234, “For each word in each string”>

Map

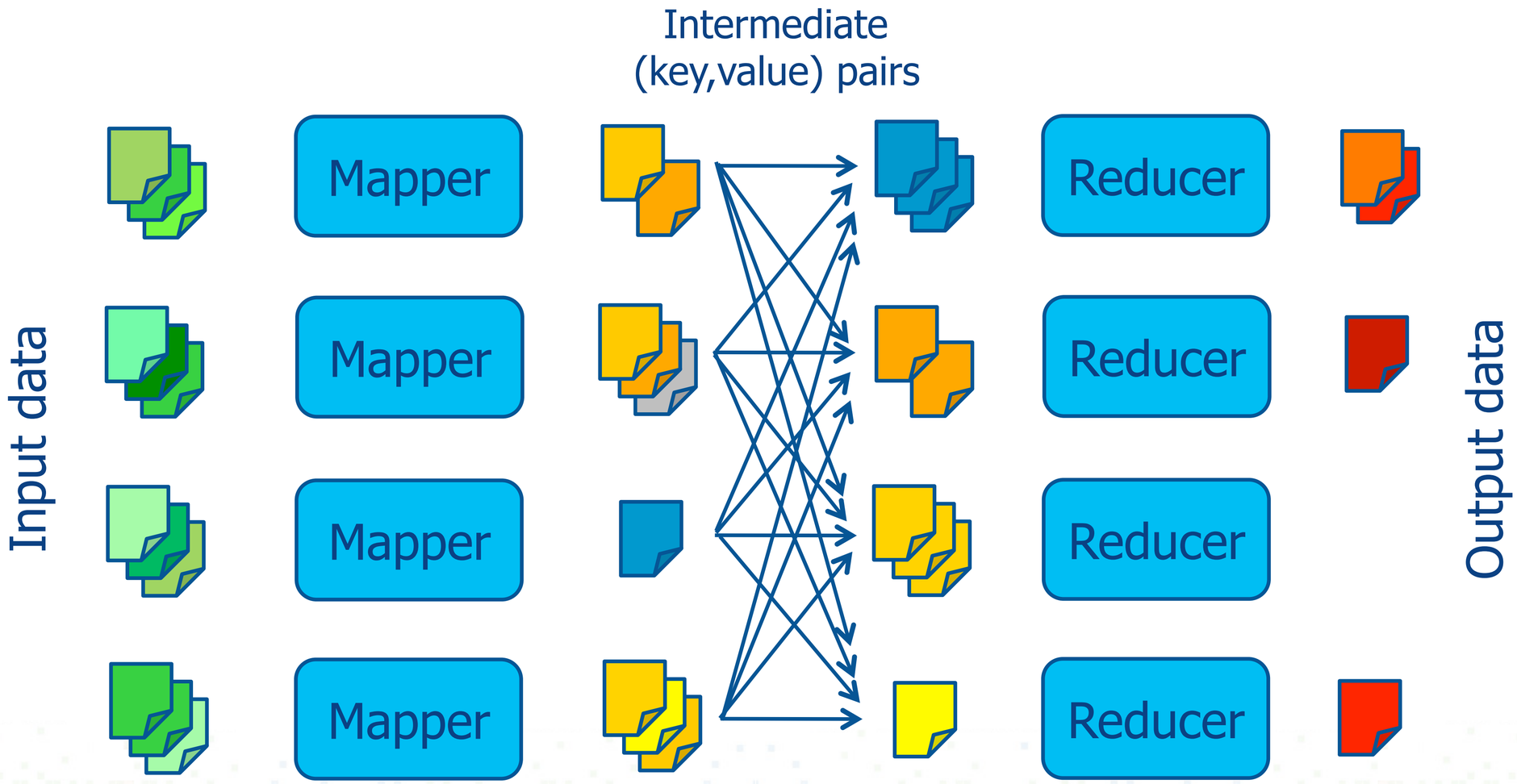
<For, 1> <each, 1> <word, 1> <in, 1> <each, 1> <string, 1>

<For, [1]> <each, [1,1]>…

Reduce

<For, 1>
<each, 2>
<word, 1>
<in, 1>
<string, 1>

MapReduce Dataflow



[*Zach Ives — UPenn*]

When to use MapReduce?

- Data parallelism problems (“embarrassingly parallel”)
  - Word count
  - Distributed grep
  - Reverse index
  - Document OCR
- Data must be splittable into chunks and records

More examples

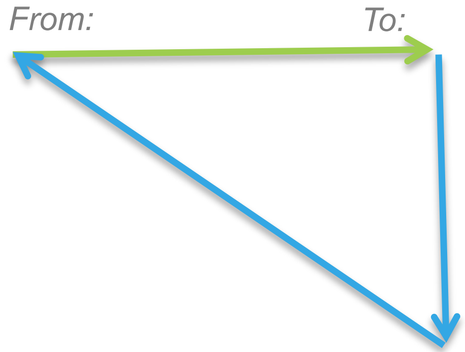
- Distributed grep – all lines matching a pattern
  - Map: filter by pattern
  - Reduce: output set
- Inverted index
  - Map: output (word,documentID)
  - Reduce: combines these into (word,[documentID])
- Document OCR:
  - Map: process (documentID:page, text lines)
  - Reduce: combine (documentID:page, [text lines])

Example: Social Triangle

- Goal: inspect emails for To:From pairings to determine social relationship of any two contacts
  - A email B and B email A ➔ A~B is reciprocal
  - A email B but B doesn't email A ➔ A~B is directed

Date: Thu, 4 May 2000 09:55:00 -0700 (PDT)
From: walt.zimmerman@enron.com
To: michael.burke@enron.com, dana.gibbs@enron.com, lori.maddox@enron.com, susan.ralph@enron.com
Subject: Update on Steve Todoroff Prosecution--CONFIDENTIAL/SUBJECT TO ATTORNEY-CLIENT PRIVILEGE
Cc: steve.duffy@enron.com, stanley.horton@enron.com, jdegeeter@velaw.com

Social Triangle: First Directed Edge



Mapper1

Maps two regular expression searches:
To: Michael, Dan, Lori, Susan
From: Walt

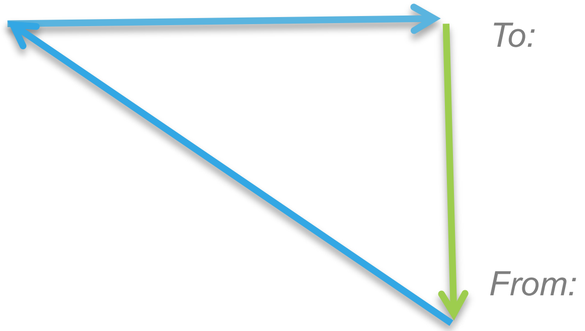
Emits the outbound directed edge of the social graph:
<Key, Value> = <Walt, [Michael, Dan, Lori, Susan]>

Reducer1

Gets the output from the mapper with different values
<Key, Value> = <Walt, [Michael, Dan, Lori, Susan]>
<Key, Value> = <Walt, [Lori, Susan, Jeff, Ken]>

Unions the values for the second directed edge:
<Key, Value> = <Walt, [Dan, Jeff, Ken, Lori, Michael, Susan]>

Social Triangle: Second Directed Edge



Mapper2

Reverses the previous Map:
To: Michael, Dan, Lori, Susan
From: Walt

Emits the inbound directed edge of the social graph:
<Key, Value> = <Susan, Walt>; <Lori, Walt>; <Dan, Walt>; etc

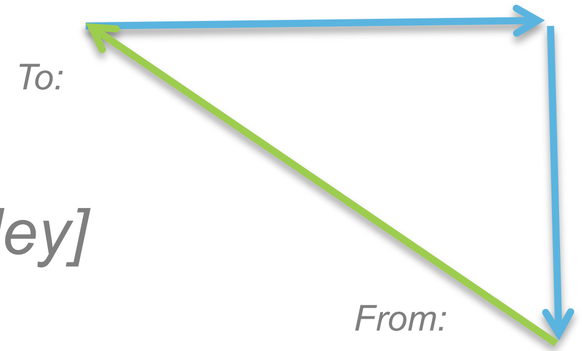
Reducer2

Gets the output from the mapper with different values
<Key, Value> = <Susan, Walt>
<Key, Value> = <Susan, Jeff>

Unions the values for the third directed edge:
<Key, Value> = <Susan, [Jeff, Ken, Walt]>

Social Triangle: Third Directed Edge

Mapper3



Join [inbound] and [outbound] lists by Key
Walt, [Jeff, Ken, Lori, Susan], [Jeff, Lori, Stanley]

Emits <Person, Person> pair with level of association:
<Key, Value> = <Walt: Jeff reciprocal>; <Walt:Stanley directed>,
etc

Reducer3

Reducer unions the output of the mappers and presents rules:
<Key, Value> = <Walt::Jeff, reciprocal>
<Key, Value> <Walt::Stanley, directed>

Designing MapReduce Algorithms

- Key decision: What should be done by **map**, and what by **reduce**?
  - **map** can do something to each individual key-value pair, but it can't look at other key-value pairs
    - Example: Filtering out key-value pairs we don't need
  - **map** can emit more than one intermediate key-value pair for each incoming key-value pair
    - Example: Incoming data is text, map produces (word,1) for each word
  - **reduce** can aggregate data; it can look at multiple values, as long as map has mapped them to the same (intermediate) key
    - Example: Count the number of words, add up the total cost, ...

Designing MapReduce Algorithms

- Need to get the intermediate format right!
  - If **reduce** needs to look at several values together, map must emit them using the same key!
- Work the algorithm bottom up
  - Look at the desired output (k3,v3), and what are needed to compute them to design your reducer and (k2,v2)
  - Then look at what are needed to compute (k2,v2), hopefully to get (k1,v1) and your mapper
- Sometime more than one MapReduce jobs are needed to finish a task

Filtering with MapReduce

- Goal: find lines/files/tuples with a particular characteristic

- Examples:
  - grep Web logs for requests to a specific domain
  - find in the Web logs the hostnames accessed by a particular IP
  - locate all the files that contain the words 'Apple' and 'Jobs'

- Generally: **map** does most of the work, utilizing one-to-many relationships to discard unqualified records. **reduce** may simply be the identity

Aggregation with MapReduce

- Goal: compute the maximum, the sum, the average, …, over a set of values

- Examples:
  - Count the number of requests to a website
  - Find the most popular domain
  - Average the number of requests per page per Web site

- Often: **map** may be simple or the identity, **reduce** does the aggregation

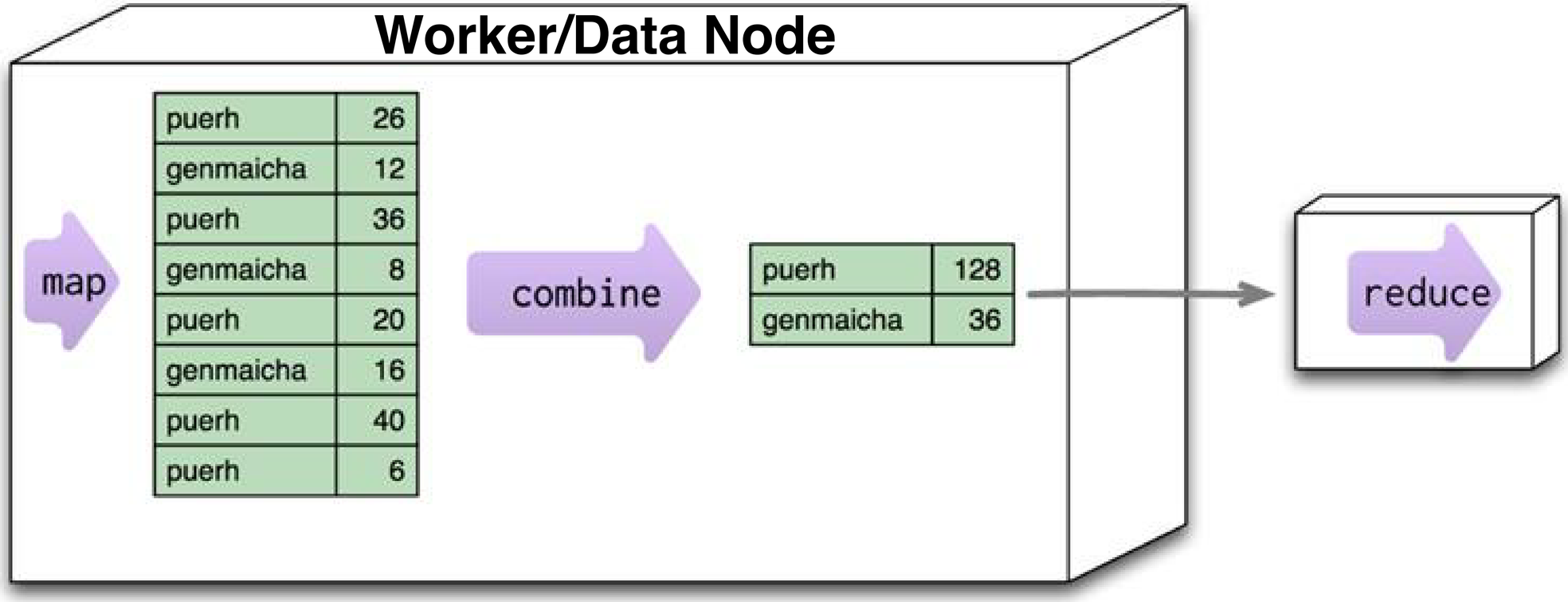
Combiners

- Certain functions can be decomposed into partial steps:
  - Can take counts of two sub-partitions, sum them up to get a complete count for the partition
  - Can take maxes of two sub-partitions, max them to get a complete max for the partition

- Multiple map jobs on the same machine may write to the same reduce key

-

Combiner Example



[Source: *Riccardo Torlone*]

Combiner Example (vs. Reducer)

dragonwell	ann
puerh	ann
puerh	brian
genmaicha	claire
puerh	david
dragonwell	ann
hojicha	ann
puerh	claire
genmaicha	claire

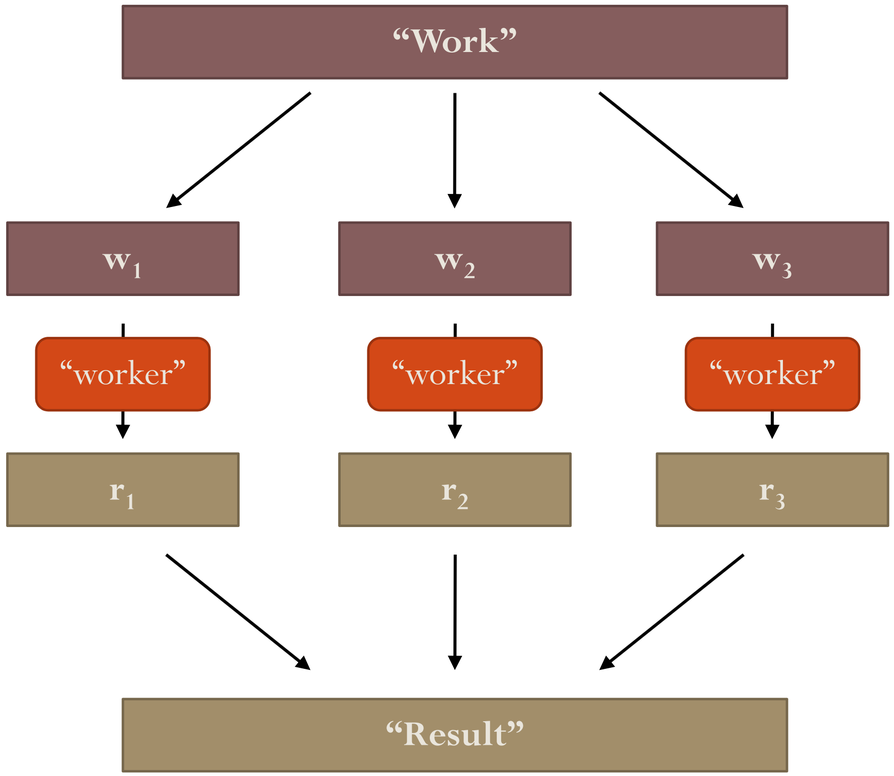
dragonwell	1
puerh	**4**
genmaicha	1
hojicha	1

Counting unique names per Last Name

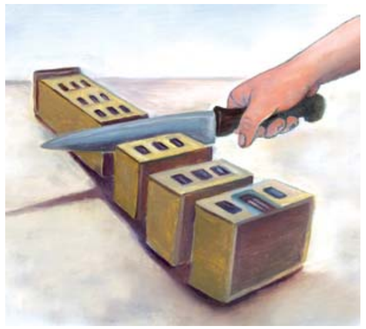
[Source: *Riccardo Torlone*]

MapReduce for Big Data

- Data parallelism — *scaling out*
  - Divide and Conquer
- Distributed computing
  - Data on different machines
- Bring compute to the data
  - Reduce transferring cost



Divide



Conquer

[Source: *Riccardo Torlone*]

What are the challenges?

- How do we distribute data across machines?
- How do we keep track of where the data are?
- How do we assign work units to machines?
- What if we have more work units than machines?
- What if a machine dies?
- Can we recover from a crash?

What are the challenges?

- How do we distribute data across machines?
- How do we keep track of where the data are?

Distributed File System

- How do we assign work units to machines?
- What if we have more work units than machines?
- What if a machine dies?
- Can we recover from a crash?

What are the challenges?

- How do we keep track of where the data are?

Distributed File System

- How do we assign work units to machines?
- What if we have more work units than machines?

Resource Manager

- What if a machine dies?
- Can we recover from a crash?

What are the challenges?

- How do we distribute data across machines?
- How do we keep track of where the data are?

Distributed File System

- How do we assign work units to machines?
- What if we have more work units than machines?

Resource Manager

- What if a machine dies?
- Can we recover from a crash?

Both

What is  ?

People use “Hadoop” to mean one of four things:

- MapReduce paradigm.
- Massive unstructured data storage on commodity hardware.

(ideas)

- Java Classes for HDFS types and MapReduce job management.
- HDFS: The Hadoop distributed file system.

(actual Hadoop)

With Hadoop, you can do MapReduce jobs quickly and efficiently.

What is  ?

People use “Hadoop” to mean one of four things:

open-source!

- MapReduce paradigm.
- Massive unstructured data storage on commodity hardware.

(ideas)

- Java Classes for HDFS types and MapReduce job management.
- HDFS: The Hadoop distributed file system.

(actual Hadoop)

With Hadoop, you can do MapReduce jobs quickly and efficiently.

…in other words

- Hadoop = HDFS + MapReduce
- Hadoop = Big Data + Analytics

Two Main Components

Storage (Big Data)

- HDFS – Hadoop Distributed File System
- Reliable, redundant, distributed file system optimized for large files

MapReduce (Analytics)

- Programming model for processing sets of data
- Mapping inputs to outputs and reducing the output of multiple Mappers to one (or a few) answer(s)

…in other words

- Hadoop = HDFS + MapReduce
- Hadoop = Big Data + Analytics

Two Main Components

Storage (Big Data)

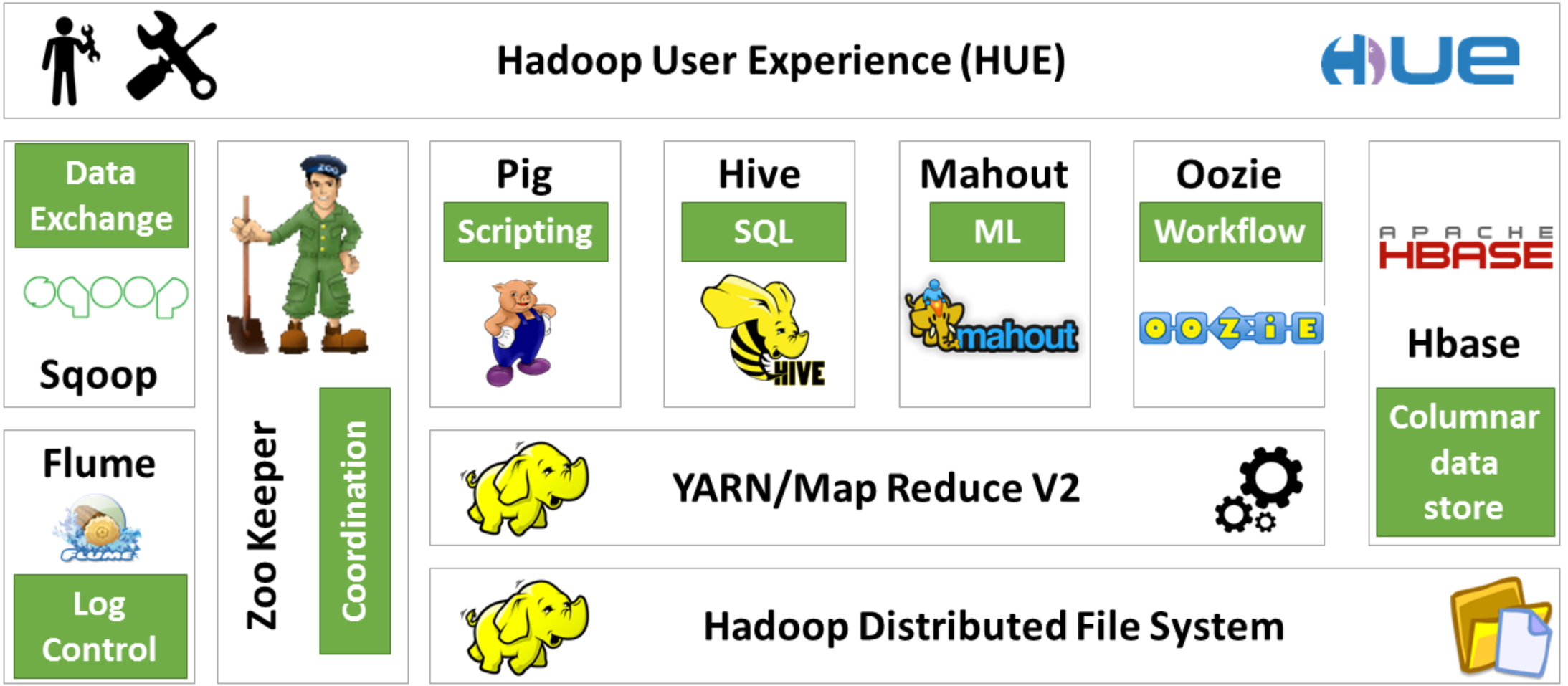
- HDFS – Hadoop Distributed File System
- Reliable, redundant, distributed file system optimized for large files

MapReduce (Analytics)

- Programming model for processing sets of data
- Mapping inputs to outputs and reducing the output of multiple Mappers to one (or a few) answer(s)

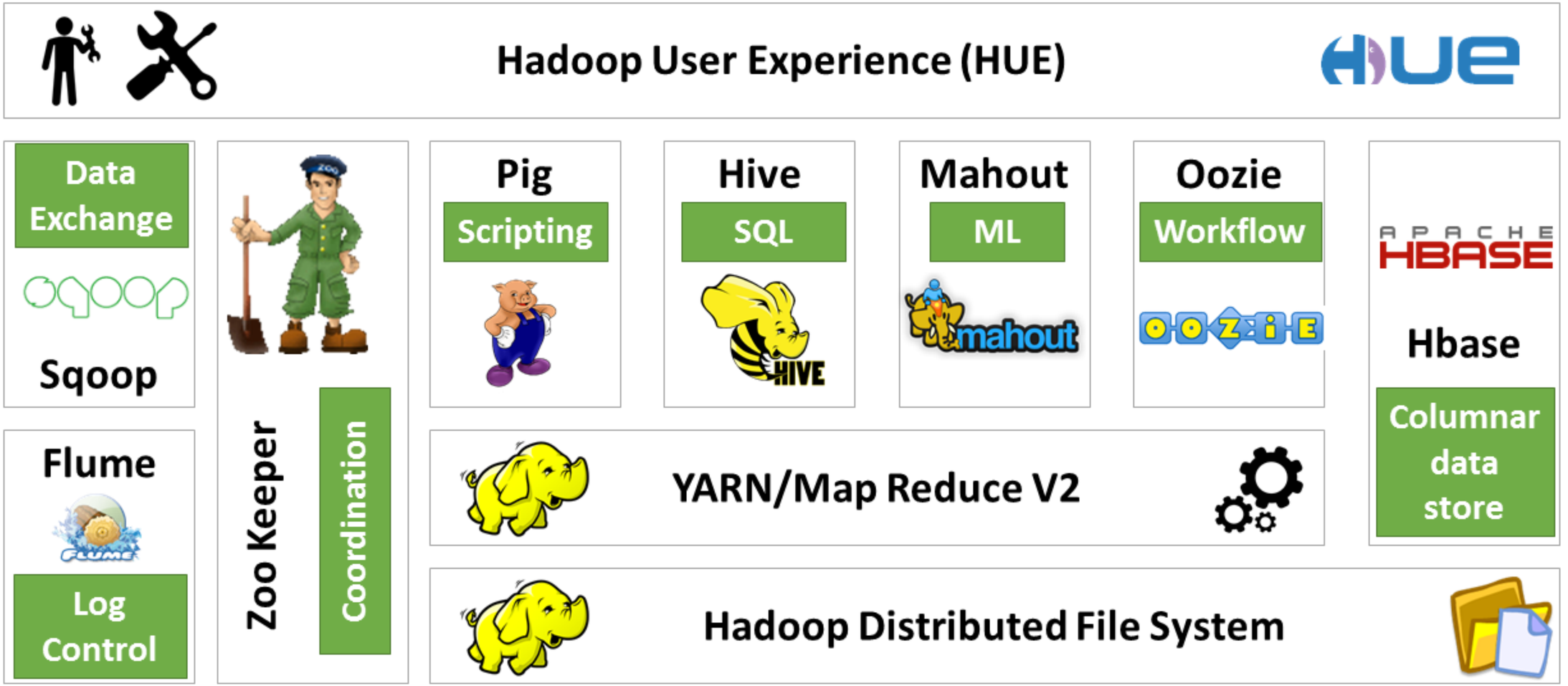
+ YARN (Yet Another Resource Negotiator)

Hadoop Ecosystem



[Source: *Banyan*]

Hadoop Ecosystem



making data center become a computer

[Source: *Banyan*]

(Hadoop) Distributed File System

- Commodity hardware
- Significant failure rates
  - Nodes can fail over time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

[Source: *Riccardo Torlone*]

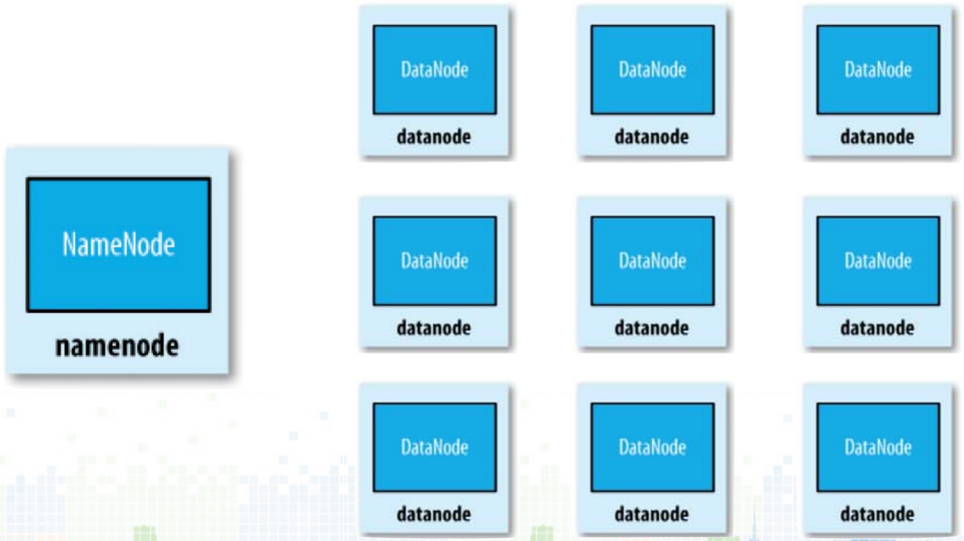
(H)DFS: Organization

- Files are divided into **chunks** (typically 64MB in size)
- Chunks are **replicated** at different compute nodes (usually 3+)
- Nodes holding copies of one chunk are located on different racks
- Chunk size and the degree of replication can be decided by the user
- A special file (the **master node**) stores, for each file, the positions of its chunks
- A directory for the file system knows where to find the master node
- Both the master node and the directory itself can be replicated
- All participants using the (H)DFS know where the directory copies are

[Source: *Riccardo Torlone*]

HDFS Implementation

- Master/slave architecture
- There are two types of nodes:
  - NameNode: a master server that manages the file system namespace and regulates access to files by clients
  - DataNode: holds data blocks, attached to storage devices
  -



[Source: *Riccardo Torlone*]

HDFS Implementation

- The NameNode:
  - Manages the filesystem tree and the metadata for all the files and directories
  - Knows the DataNodes on which all the blocks for a given file are located
- The DataNode:
  - Just store and retrieve the blocks when they are told to (by clients or the NameNode). Internally, a file is split into one or more blocks and these blocks are stored in a set of data nodes
- Without the NameNode HDFS cannot be used
  - The NameNode machine is a single point of failure for an HDFS cluster
- It is important to make the namenode resilient to failure

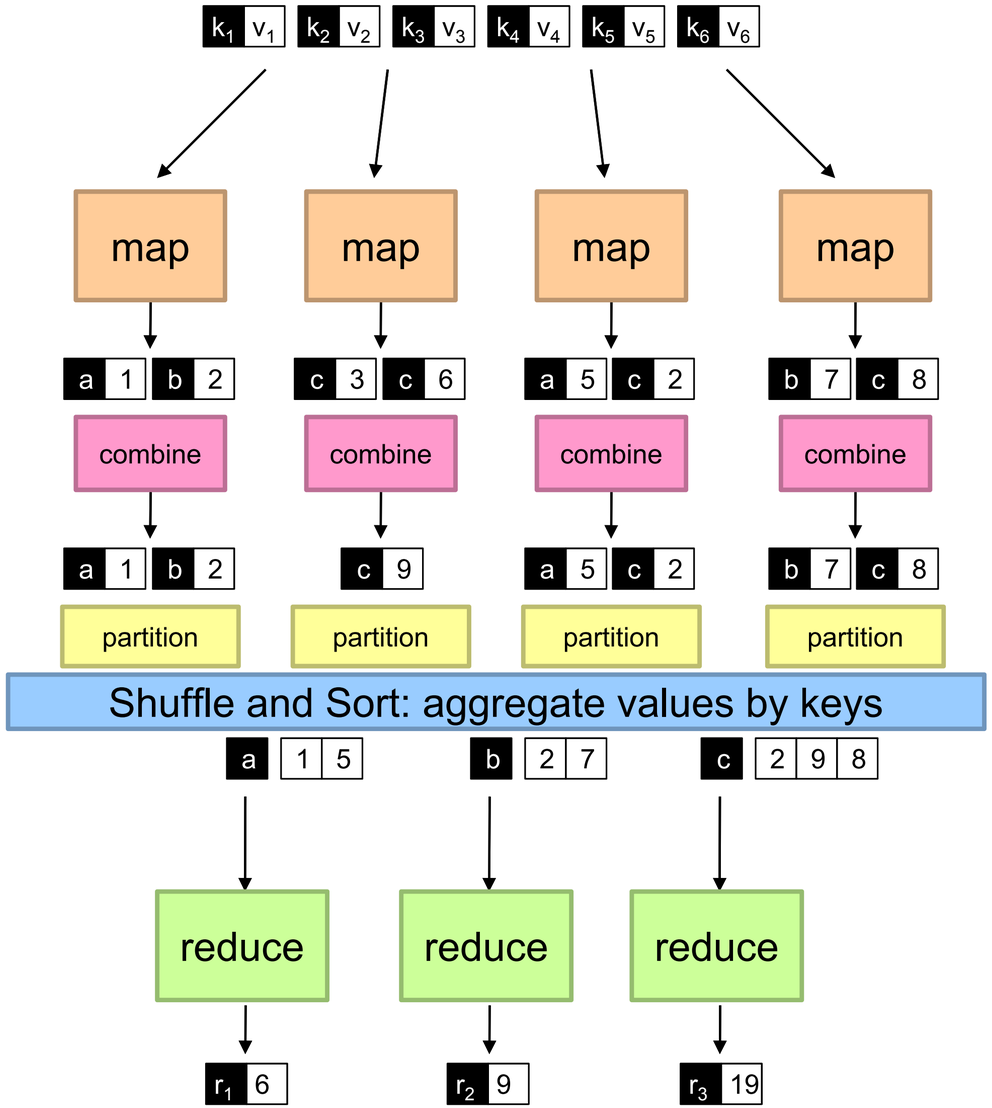
[Source: *Riccardo Torlone*]

HDFS I/O

- An application client wishing to read a file (or a portion thereof) must first contact the NameNode to determine where the actual data is stored
- In response to the client request the NameNode returns:
  - the relevant block ids
  - the location where the blocks are held (i.e., which DataNodes)
- The client then contacts the DataNodes to retrieve the data
- Blocks are themselves stored on standard single-machine file systems
  - HDFS lies on top of the standard OS stack
- Important feature of the design:
  - data is never moved through the NameNode
  - all data transfer occurs directly between clients and DataNodes
  - communications with the NameNode only involves transfer of metadata

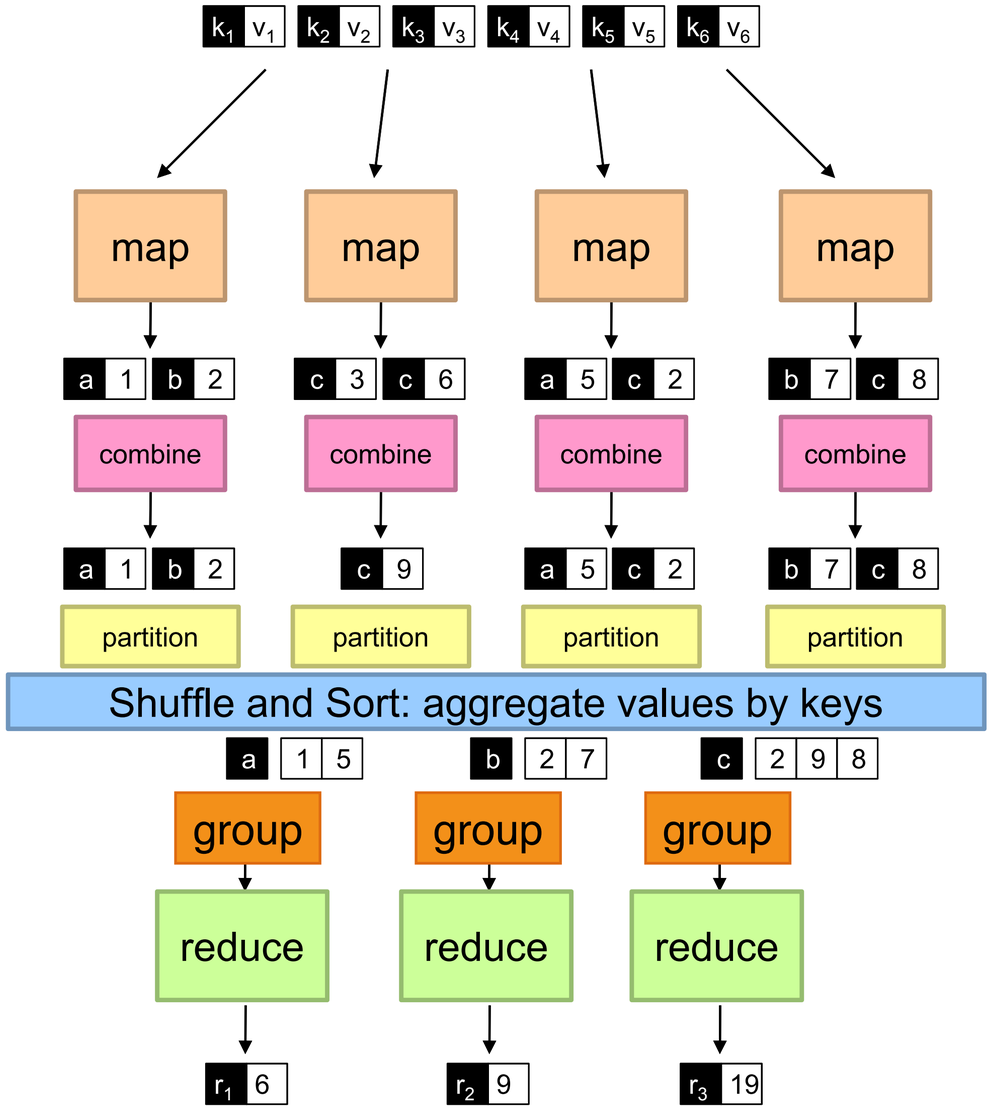
[Source: *Riccardo Torlone*]

Hadoop and MapReduce



[Source: *Jimmy Lin*]

Hadoop and MapReduce

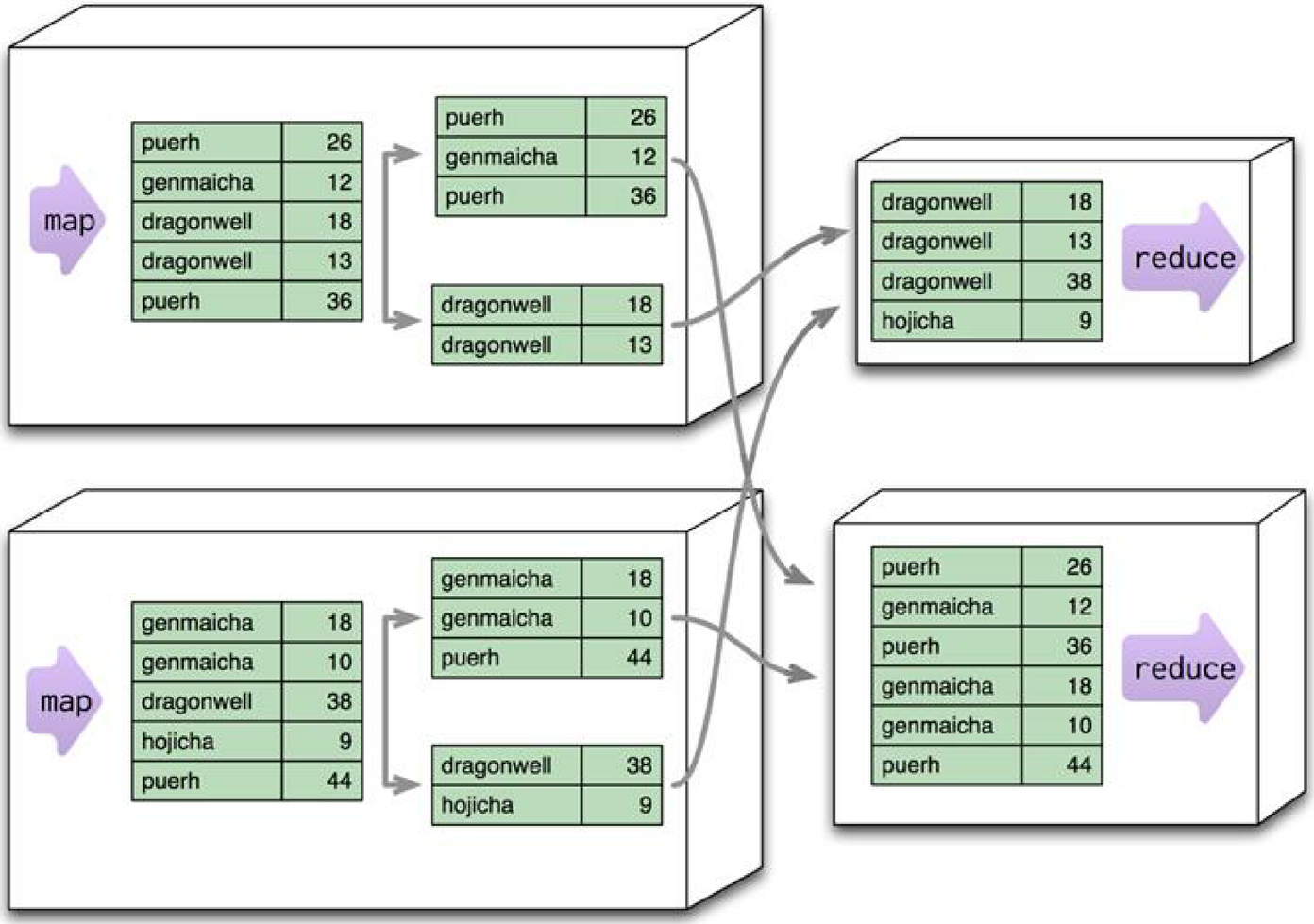


[Source: *Jimmy Lin*]

Partitioners

- We can also specify a partitioner that:
  - divides up the intermediate key space
  - assigns intermediate key-value pairs to reducers
  - n partitions ➔ n reducers (n can be specified by users)
- The simplest partitioner assigns approximately the same number of keys to each reducer
- Partitioner only considers the key and ignores the value
- Optionally, users can specify their own hash function or other method for assigning keys to Reduce tasks.
- However, whatever algorithm is used, each key is assigned to one and only one Reducer

Partitioners Example



Hadoop Runtime Tasks

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Get data to the workers
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS

Locality Enforcement

- Don’t move data to workers... move workers to the data!
- Store data on the local disks of nodes in the cluster
- ***Start up the workers on the node that has the data local***
- The tasks are created from the input splits in the shared file system
  - 1 map per split + N reduces determined by the configuration
- If this is not possible (e.g., a node is already running too many tasks)
  - new tasks will be started elsewhere
  - the necessary data will be streamed over the network
- Optimization
  - prefer nodes that are on the same rack in the datacenter as the node holding the relevant data block
  - inter-rack bandwidth is significantly less than intra-rack bandwidth!

Shared-nothing Architecture

- Coordinating the process in a distributed computation is hard
  - How to distribute the load over the available nodes?
  - How to serialize the data for the transmission?
  - How to handle an unresponsive process?
- MapReduce's shared-nothing architecture means that tasks have no dependence on one other
- Programmers do not worry about the distributed computation issues
- They need only to write two functions: the Map and Reduce functions

Fault-Tolerance

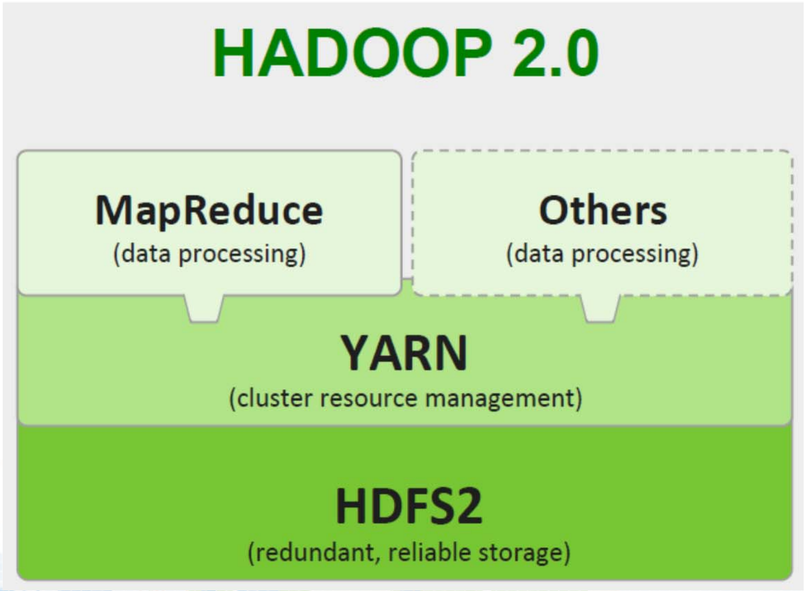
- The worst thing that can happen is that the compute node executing the Master fails
  - In this case, the entire map-reduce job must be restarted.
- Other failures will be detected and managed by the Master, and the map-reduce job will complete eventually.
- Failure of a worker
  - the tasks assigned to this Worker will have to be redone
  - the Master:
    - sets the status of each failed tasks to idle
    - reschedules them on a Worker when one becomes available

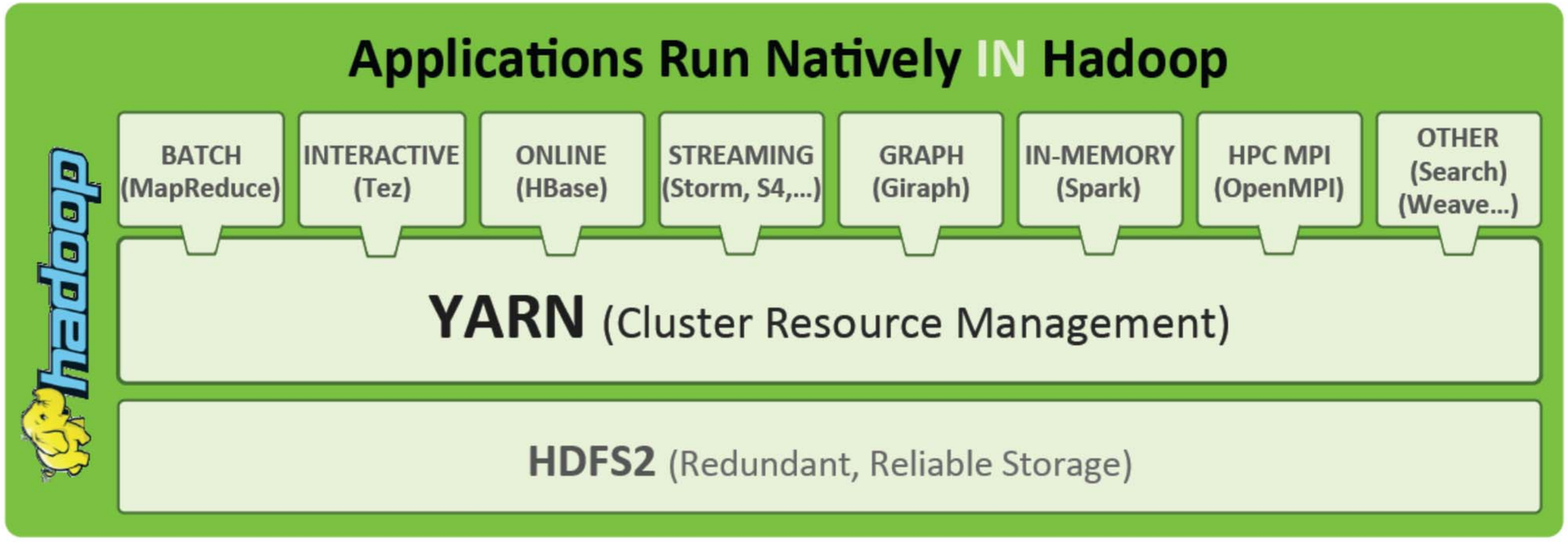
YARN

- MapReduce job lifecycle is handled by YARN — a resource negotiator
  - beyond batch execution

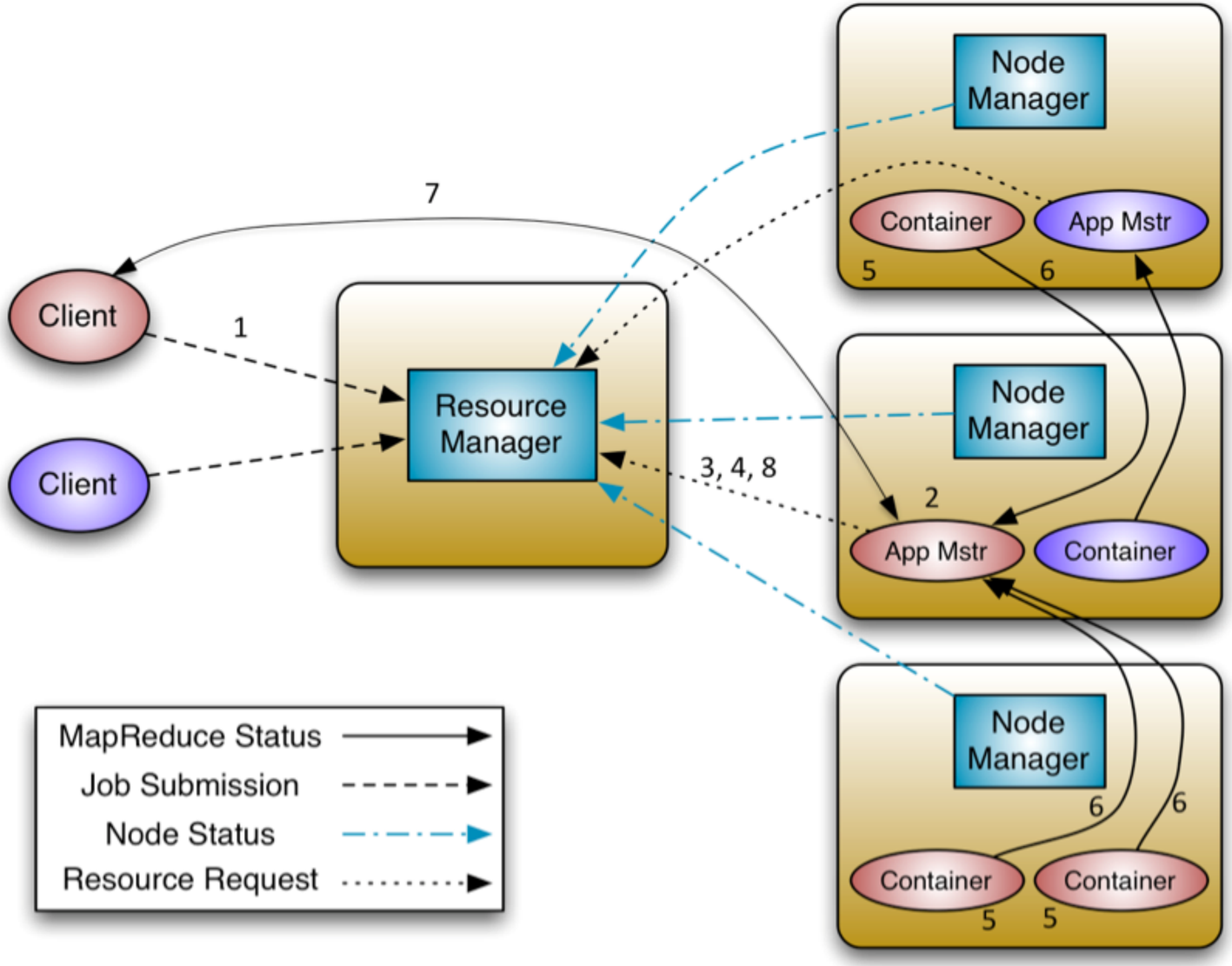
Multi Purpose Platform

Batch, Interactive, Online, Streaming, …

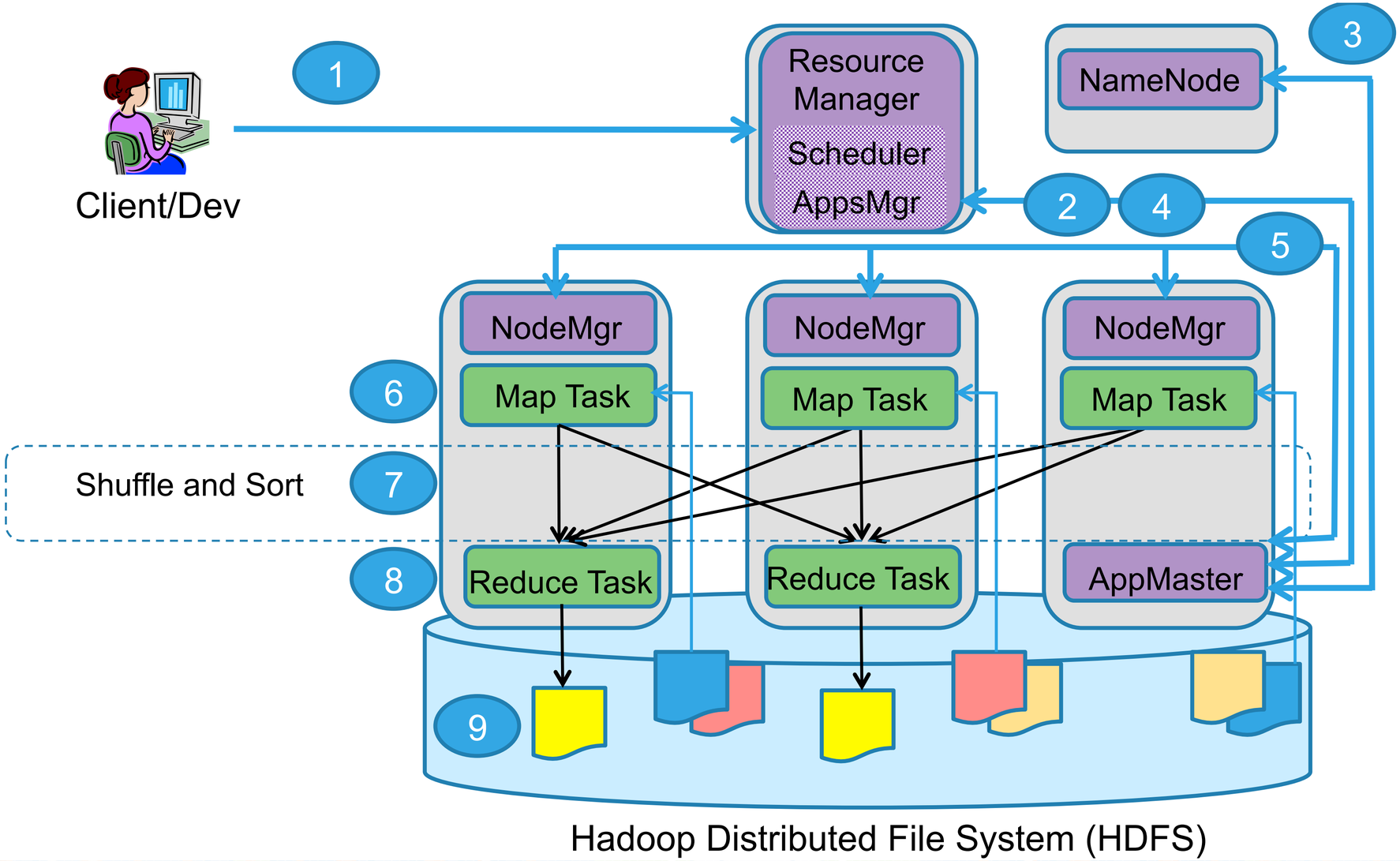




YARN — Job Running Mechanism



Putting It All Together: Hadoop (YARN)



Hadoop Operation Modes

- Java MapReduce Mode
  - Write Mapper, Combiner, Reducer functions in Java using Hadoop Java APIs
  - Read records one at a time
- Streaming Mode
  - Uses *nix pipes and standard input and output streams!
  - Any language (Python, Ruby, C, Perl, Tcl/Tk, etc.)
  - Input can be a line at a time, or a stream at a time

Hadoop Streaming Jobs

- Reducers receive independent key/value pairs (not lists of values)
- The mapper and reducer must be in executable form
  - Complete Python programs (.py file) that get sent to worker nodes
- Some features are not exposed, e.g. custom format and/or partitioner
- Input and output are handled through **stdin** and **stdout**
  - Might not perform as well/efficient as native codes

Hadoop Streaming Example

	Script to invoke Hadoop
1	`Hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-streaming.jar \`
2	`-input input/ \ # relative to HDFS`
3	`-output output \ # relative to HDFS`
4	`-mapper mapper.py \`
5	`-reducer reducer.py \`
6	`-file scripts/mapper.py \`
7	`-file scripts/reducer.py`

Example Mapper

Code	Comments
import sys	
for line in sys.stdin	# input comes from STDIN
line = line.strip()	# strip leading/trailing whitespace
words = line.split()	# split line based on whitespace
for word in words: print '%s\t %s %(word, 1)	# for each word in the collection words # write word and count of "1", tab delimited

Example Reducer

Code	Code, continued
cur_count = 0	if cur_word == word:
cur_word = None	cur_count += count
for line in sys.stdin	else:
line = line.strip()	if cur_word:
word, count = line.split("\t",1)	print '%s\t%s' (cur_word, cur_count)
try	cur_count, cur_word = (count, word)
count = int(count)	if cur_word == word:
except ValueError:	print '%s\t%s' %(cur_word, cur_count)
continue	

Hadoop Streaming

- Operate on streaming “chunks” of data
  - By default only reads and writes lines of text
- **“group-by-key” has to be done manually**
- Allow for Python **scripts** for mappers, reducers and combiners
- Can control the number of reducers and mappers
- Intermediate keys are split by tabs ‘\t’ by default
- More customizations (e.g. partitioner) have to be done in Java

Hadoop Streaming Mapper

```
#!/usr/bin/env python
import sys, time

def parseRecords():
    for line in sys.stdin:
        line = line.strip('\n')
        yield line.split()

def mapper():
    for words in parseRecords():
        for w in words:
            print '%s\t%s' % (w,1)

if __name__=='__main__':
    mapper()
```

Hadoop Streaming Reducer

```
#!/usr/bin/env python
import itertools, operator, sys

def parsePairs():
    for line in sys.stdin:
        yield tuple(line.strip('\n').split('\t'))

def reducer():
    for key, pairs in itertools.groupby(parsePairs(),
                                        operator.itemgetter(0)):
        count = sum(int(i[1]) for i in pairs)
        print '%s\t%s' % (key, count)

if __name__=='__main__':
    reducer()
```

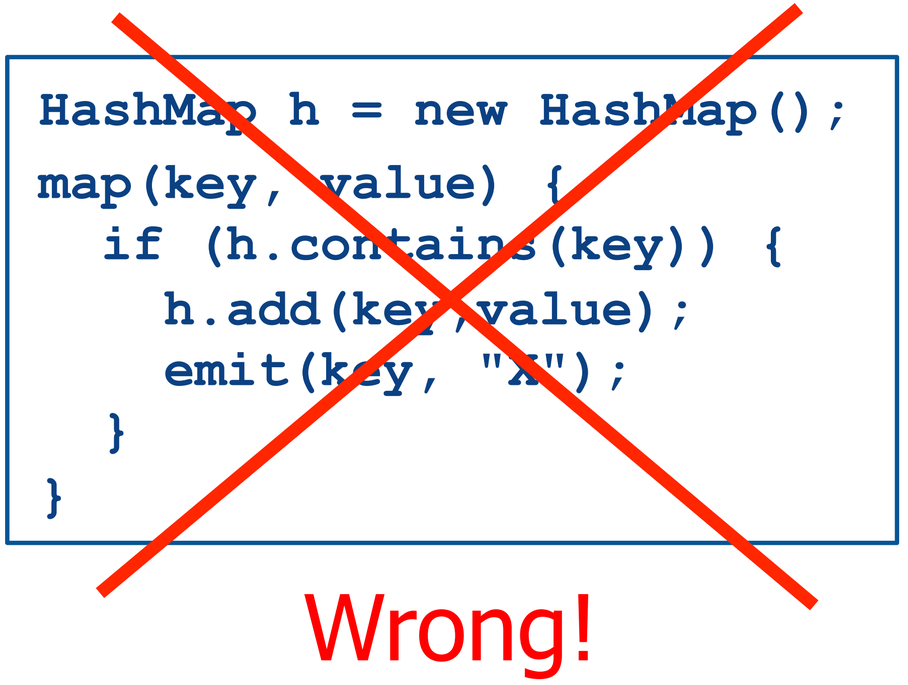
How to debug Hadoop program?

- Using the mapreduce.py package for testing algorithm correctness
- Testing your mapper.py and reducer.py locally for streaming logics

```
cat input.txt | ./mapper.py | sort -k 1,1 | ./reducer.py > output.txt
```

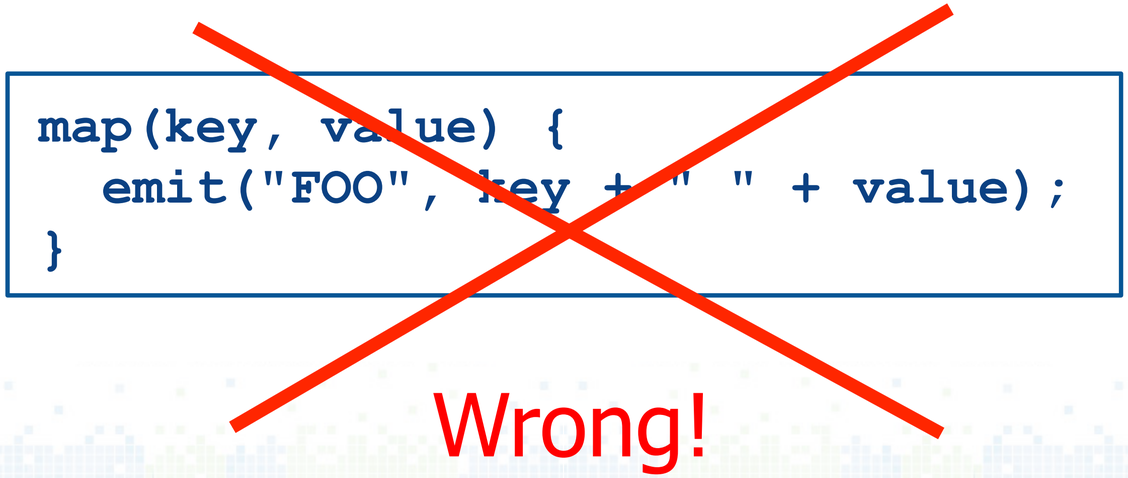
Common Mistakes to Avoid

- Mapper and reducer should be **stateless**
  - Don't use static variables - after map + reduce return, they should remember nothing about the processed data!
  - Reason: No guarantees about which key-value pairs will be processed by which workers!



Common Mistakes to Avoid

- Mapper must not map **too much data** to the **same key**
  - In particular, don't map everything to the same key!! Otherwise the reduce worker will be overwhelmed!
- It's okay if some reduce workers have more work than others



Thanks!

- Questions?